HAL
archives-ouvertes.fr

Lazy Solid Texture Synthesis

Yue Dong, Sylvain Lefebvre, Xin Tong, George Drettakis

To cite this version:

Yue Dong, Sylvain Lefebvre, Xin Tong, George Drettakis. Lazy Solid Texture Synthesis. 19th Eurographics Symposium on Rendering, Jun 2008, Sarajevo, Bosnia and Herzegovina. inria-00606812

HAL Id: inria-00606812
https://hal.inria.fr/inria-00606812

Submitted on 18 Jul 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Eurographics Symposium on Rendering 2008
Steve Marschner and Michael Wimmer
(Guest Editors)

Volume 27 (2008), Number 4

Lazy Solid Texture Synthesis

Yue Dong[1,2], Sylvain Lefebvre[3], Xin Tong[1] and George Drettakis[3]

[1] Microsoft Research Asia [2] Tsinghua University [3] REVES / INRIA Sophia-Antipolis

Abstract

Existing solid texture synthesis algorithms generate a full volume of color content from a set of 2D example images. We introduce a new algorithm with the unique ability to restrict synthesis to a subset of the voxels, while enforcing spatial determinism. This is especially useful when texturing objects, since only a thick layer around the surface needs to be synthesized. A major difficulty lies in reducing the dependency chain of neighborhood matching, so that each voxel only depends on a small number of other voxels.
Our key idea is to synthesize a volume from a set of pre-computed 3D candidates, each being a triple of interleaved 2D neighborhoods. We present an efficient algorithm to carefully select in a pre-process only those candidates forming consistent *triples. This significantly reduces the search space during subsequent synthesis. The result is a new parallel, spatially deterministic solid texture synthesis algorithm which runs efficiently on the GPU.*
Our approach generates high resolution solid textures on surfaces within seconds. Memory usage and synthesis time only depend on the output textured surface area. The GPU implementation of our method rapidly synthesizes new textures for the surfaces appearing when interactively breaking or cutting objects.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism – Color, shading, shadowing, and texture;

1. Introduction

Texture mapping greatly improves the visual appearance of object surfaces without additional geometry. This is typically achieved by parameterizing the surface in the plane, and by defining color content in a flat image. However, one has to compensate for the distortion and discontinuities introduced by the mapping when creating the texture content.

Instead, *solid textures* define the texture content directly in 3D, hence removing the need for a planar parameterization and producing the unique feeling that the object has been carved out of a block of matter. This was first proposed in the context of procedural texturing [EMP*94]: The color in a point of the volume is *computed* by a small algorithm. This uses very little memory since only the algorithm is stored. However, the most desirable property is that the color computation at any point is independent from other points. This *local* evaluation allows the restriction of color computations to the visible surface points. Unfortunately, procedures can only be defined for a limited set of materials, such as marble, wood and cellular patterns.

Recent work on texture synthesis from example alleviates this problem [Wei02, QY07, KFCO*07]: A 3D volume is automatically generated from 2D images provided as examples. However, existing approaches typically solve a *global* problem: The dependency chain to compute the color of any voxel potentially involves *all* other voxels. A direct consequence is that synthesis cannot be restricted to a subset of the volume. On surfaces, storage is orders of magnitude more expensive than a 2D texture map of similar resolution. In addition to being wasteful in terms of memory, the inherently cubic size of the volume data results in high computational complexity, when ideally the complexity in time and space should only depend on the surface area to be textured.

Our new synthesis algorithm fills the gap between procedural textures, that are limited but can be evaluated at display time, and solid synthesis, which requires the precomputation and storage of an entire volume. Our approach starts from 2D images and synthesizes a solid texture on a surface, as if it was carved out of a volume of matter. However, only the required parts of the volume are effectively synthesized. To determine the color of a voxel, our method only has to compute the color of a small number of surround-

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.
Published by Blackwell Publishing, 9600 Garsington Road, Oxford OX4 2DQ, UK and 350 Main Street, Malden, MA 02148, USA.

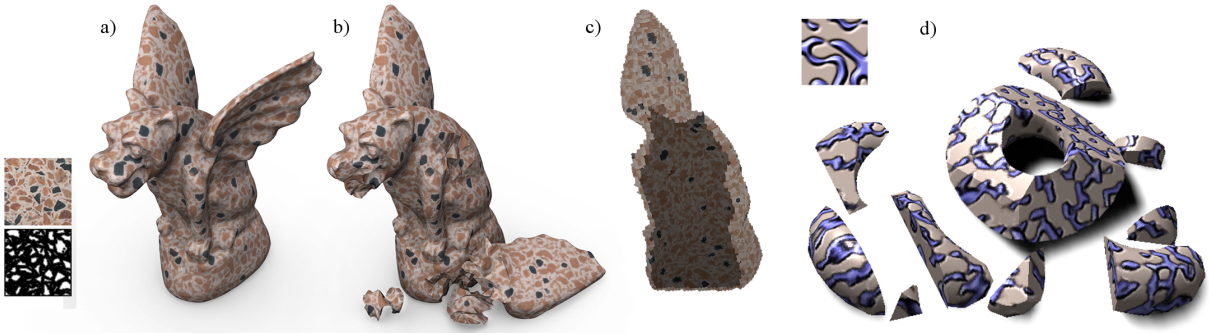


Figure 1: *Lazy solid synthesis starts from 2D images and synthesizes a solid texture on a surface. Only the required parts of the volume are effectively synthesized.* a) *A gargoyle textured by a* 1024^3 *volume.* b) *Thanks to spatial determinism, the object may be broken and the interior textured in a consistent manner.* c) 0.42% *of the voxels are synthesized by our algorithm, requiring only* 5.4 *seconds for synthesis and* 17.9*MB of storage, instead of* 3*GB for the full volume.* d) *Our approach enables interactive cutting of objects with on demand synthesis for the appearing surfaces. New textures are synthesized in about 8 milliseconds.*

ing voxels. The result for an entire surface is computed once, usually in a few seconds, and stored for later display. Our GPU implementation is fast enough to synthesize textures on demand, in a few milliseconds, for surfaces appearing when interactively cutting or breaking objects.

The key idea making our approach possible is to pre-compute a small number of carefully selected *3D candidates*, later used in our solid synthesis algorithm. Each candidate is formed by interleaving three well-chosen 2D neighborhoods from the example images. Our pre-computed 3D candidates improve synthesis efficiency and significantly reduce the dependency chain required to compute voxel colors. This allows us to develop our new parallel, spatially deterministic solid texture synthesis algorithm.

The result is a *lazy* solid synthesis scheme, only computing colors in voxels actually being used (see Fig. 1). High resolution solid textures are applied to objects in seconds, at low memory cost. Our GPU implementation is fast enough to synthesize textures on demand, enabling interactive cutting and breaking of objects. Our synthesis scheme is spatially deterministic: The same color is always generated at the same location in space. Hence, the textures synthesized for appearing surfaces remain consistent.

2. Previous Work

Background on texture synthesis from example. Planar textures are typically synthesized from an example image (or *exemplar*) by pasting together small pixel neighborhoods [EL99, WL00, KEBK05] or entire patches of texture content [EF01, KSE*03]. The parallel per-pixel synthesis algorithms of [WL03, LH05] are of particular interest to us: They enable efficient synthesis of subsets of an image while enforcing spatial determinism. This is exactly the property we want to obtain for solid synthesis. However, the extension to 3D is not straightforward: No full volume is available as input. Even if such a volume were available, its cubic size would imply both a large increase in computation and memory consumption.

Most per-pixel algorithms follow a same approach: In each pixel, a small neighborhood representing its current surrounding is extracted. The exemplar is then searched for the pixel with the most similar neighborhood, and the pixel color is copied to the output. To achieve fast synthesis, recent schemes make use of *k-coherent candidates* [Ash01, TZL*02]. The key idea is to pre-compute a set of candidates in each exemplar pixel: Typically the coordinates of the k pixels having the most similar neighborhood. During synthesis, the search for a best matching neighborhood is limited to candidates in the pre-computed sets of already synthesized pixels. This provides a significant speed-up. Of course, computing good candidates is key to fast, high quality synthesis.

Solid (3D) texture synthesis from example. We now focus on the existing approaches for volume texture synthesis from example. For a detailed overview of the topic we also invite the reader to refer to [DG01].

The pyramid histogram matching of [HB95] and the spectral analysis methods of [GD95, GD96] pioneered the work on solid texture synthesis from example. The former reproduces global statistics of the 2D example images in the volume, while the latter two create a procedural solid texture from the spectral analysis of multiple images. [QY07] synthesizes a volume by capturing the co-occurrences of grayscale levels in the neighborhoods of 2D images. [JDR04] proposed a solid synthesis method targeted at aggregates of particles, whose distribution and shape is analyzed from an input image. [ONOI04] adapted constrained 2D synthesis [HJO*01] to illustrate object interiors. Spatial determinism is however not enforced and seams appear at transitions between different cuts.

[Wei02] first adapted 2D neighborhood matching syn-

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.

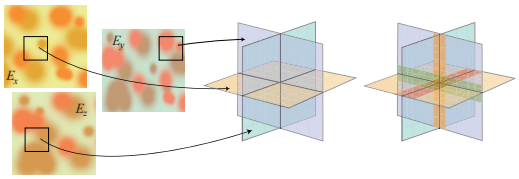


Figure 2: Left: *The three exemplars E_x, E_y, E_z and a corresponding 3-neighborhood.* Right: *the crossbar defined by the 3-neighborhood.*

thesis schemes to 3D volumes. The key idea is to consider three 2D exemplars, one in each direction. In each pixel of the output volume (*voxel*), three interleaved 2D neighborhoods are extracted (see Fig. 2). The best matches are found independently in each of the three exemplars. A local optimization step then attempts to converge toward the best color for all three directions. [KFCO*07] relies on similar interleaved neighborhoods, but uses a global optimization approach [KEBK05]. A histogram matching step is also introduced, further improving the result. Both these approaches search for best matching neighborhoods independently in each direction. Consequently, a large number of iterations are required before the algorithm stabilizes into a satisfactory result.

While these approaches produce impressive results, none is appropriate for fast synthesis of subsets of the volume: they either rely on sequential generation or global optimization. In both cases, the dependency chain in the computations implies that a complete volume has to be computed, even if the user is only interested in a small subset of the voxels.

3. Overview and terminology

Our goal is to generate high-quality, high-resolution solid textures given three 2D exemplars - often the same image repeated thrice. Since we target texturing of surfaces, we want our approach to allow very fast synthesis of subsets of the volume.

A first idea would be to revisit the solid synthesis approach of [Wei02], adapt it to be parallel and use the 2D k-coherent candidates mechanism to achieve a significant speed-up. However, it is important to realize that 2D candidates are not likely to be meaningful for 3D synthesis: Each candidate will represent a correct 2D neighborhood in its image, but once put together they are likely to introduce color inconsistencies. This will both reduce synthesis quality, and require a long time for the iterative synthesis process to produce good results. As many iterations of neighborhood matching will be required, the dependency chain to compute the color of a voxel will involve a large number of other voxels, making subset synthesis impractical. For a detailed comparison between this approach and ours, please refer to Sec.6.4.

Instead, our novel scheme pre-computes 3D candidates given three 2D example images. Each candidate is made of three interleaved 2D neighborhoods, and is carefully selected to provide both quality and speed during synthesis (Sec. 4). This is done as a pre-computation, and only once for a given set of three 2D exemplars. The candidates are later used during the neighborhood matching step of our parallel solid synthesis scheme (Sec. 5). Our algorithm perform multi-resolution synthesis in a sparse volume pyramid, only synthesizing the small subset of the voxels necessary to texture the surface. Synthesis is performed within seconds, and the result is stored for display. Our GPU implementation generates textures on demand, in a few milliseconds, for instance for new surfaces appearing when interactively cutting or breaking objects (Sec. 6).

Terminology. We now introduce some terminology. We refer to *pixels* in 2D images, while the term *voxel* is used for pixels located in a volume. In this work we consider neighborhoods made of three interleaved 2D neighborhoods: Three $N \times N$ 2D neighborhoods embedded in three orthogonal planes and meeting at their center (see Fig. 2 (middle)). In our implementation we choose $N = 5$, which provides both good quality and performance. We refer to these triples of 2D neighborhoods as *3-neighborhoods*, and name them *3D candidates* after selection. We define the *crossbar* to be the set of pixels which are contained in more than one 2D neighborhood (Fig. 2 (right)).

E_x, E_y, E_z are the three 2D exemplars corresponding to each one of the three orthogonal planes (Fig. 2 (left)). The term *triple* indicates a triple of 2D coordinates, defining a 3-neighbourhood: Each coordinate is the center of a 2D neighborhood in the corresponding exemplar.

Our algorithm performs multi-resolution synthesis in a volume pyramid, noted V. Each level is designated by V^l where $l \in [O...L]$ and L is the maximum (finest) resolution.

4. 3D candidates selection

In each pixel of each exemplar we compute a small candidate set of 3-neighborhoods represented as coordinate triples. These sets are used during synthesis as candidates for best matching neighborhoods. Hence, they must capture the appearance of the 3D neighborhoods implicitly described by the input images.

Given the exemplars E_x, E_y, E_z, the search space for possible candidates is huge: it contains all possible triples of coordinates. The key novelty of our approach is to drastically prune the search space *prior* to synthesis. The major difficulty is that we cannot explicitly test whether a candidate triple forms a neighborhood representative of the not-yet-synthesized volume. This information is only *implicitly* given by the input images.

We hence propose to select candidate triples following

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.

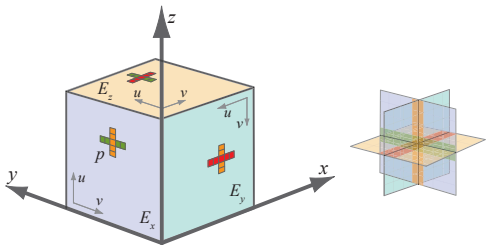


Figure 3: *Each candidate consists of three exemplar coordinates defining a 3-neighborhood. Consistent triples have low color differences along the crossbar. We seek to minimize the color differences along the three pairs of pixel strips, shown here with the same color.*

two important properties: First, a good triple must have matching colors along the crossbar of the 3-neighborhood. This provides an easy way to only select triples with good *color consistency* (see Sec. 4.1). The second property is less obvious. A major step forward in 2D texture synthesis speed and quality was achieved by giving a higher priority to candidates likely to form coherent patches from the example image [Ash01, HJO*01]. This notion is however not trivial to extend to 3D, as three exemplars are interleaved. Candidates providing coherence in one exemplar are not likely to provide coherence in the other two. Here, our approach of forming candidates prior to synthesis gives us a crucial advantage: We are able to consider coherence across *all three exemplars*, keeping only those triples likely to form coherent patches with other neighboring candidates *in all three directions* (see Sec. 4.2).

Since our synthesis algorithm is multi-resolution, we first compute an image pyramid of each 2D exemplar and apply the candidate set construction independently on each level of the pyramid of each exemplar. For clarity, the following description is for a single level.

4.1. Color consistency

Our first observation comes from the fact that a suitable candidate should be consistent across the crossbar. We use this observation to build first sets of potential candidates in each pixel of each exemplar.

As illustrated in Fig. 3, we seek to minimize the color disparity between the lines shared by interleaved 2D neighborhoods. We compute a L^2 color difference between pairs of 1-dimensional "strips" of pixels (i.e., a Nx1 or 1xN vector) from the appropriate exemplars (E_x, E_y, or E_z). The sum of color differences for the three pairs of pixel strips defines our crossbar error *CB* for any candidate triple.

In each pixel of each exemplar, we form triples using the pixel itself and two neighborhoods from the other two exemplars. We select the triples producing the smallest crossbar error. For efficiency, we approximate this process first by separately extracting the S most-similar pixel strips from each of the two other exemplars, using the ANN library [MA97]. For the example of Fig. 3, assuming we are computing a candidate set for p in E_x, we would first find in E_y the S pixel strips best matching the orange line from E_x, and in E_z the S pixel strips best matching the green line from E_x. We then produce all possible S^2 triples - using the current pixel as the third coordinate - and order them according to the crossbar error *CB*. In our results, we keep the 100 best triples and typically use a value of $S = 65$, experimentally chosen to not miss any good triple.

4.2. Triples of coherent candidates

Color consistency is only a necessary condition and many uninteresting candidate triples may be selected. As a consequence, if we directly use these candidates our algorithm will be inefficient as many will be always rejected.

After constructing candidates based on color consistency, we obtain a set of candidate triples at each pixel of each exemplar. Our key idea is to check whether a candidate may form coherent patches in *all* directions with candidates from neighboring pixels. This is in fact a simple test. We consider each coordinate within a candidate triple and verify that at least one candidate from a neighboring pixel has a continuous coordinate. Fig. 4 illustrates this idea for pixels in E_x. We only keep candidates finding continuous coordinates for all three entries of the triple. Note that one is trivially true, i.e. by definition neighboring candidates in E_x have a continuous coordinate in E_x.

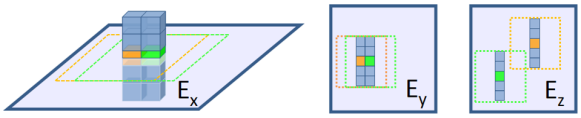


Figure 4: *Two candidates of neighboring pixels in E_x. Each candidate is a triple with coordinates in E_x, E_y and E_z. The first triple is shown in orange, the second in green. Notice how the coordinates of the candidates in E_y are contiguous: Along both vertical pixel strips, the candidates form a coherent patch from E_y. This is not the case in E_z. Note that the orange and green triples will only be kept if another neighboring triple with a contiguous coordinate in E_z is found.*

To formalize this notion, let us only consider exemplar E_x without loss of generality. We note ${}^xC$ (resp. ${}^yC$, ${}^zC$) the set of candidates for exemplar E_x (resp. E_y, E_z). ${}^xC^k[p]$ is the k-*th* candidate triple for pixel p in E_x. We note ${}^xC^k[p]_y$ and ${}^xC^k[p]_z$ the coordinates in respectively exemplar E_y and E_z for the candidate triple.

In a given pixel p, we iteratively update the set of candidates as:

$$ {}^xC_{i+1}[p] = \left\{ c \in {}^xC_i[p] : \exists k_1, k_2, |\delta_1| = 1, |\delta_2| = 1 \text{ s.t.} \left[\begin{array}{l} |{}^xC_i^{k_1}[p+\delta_1]_y - c_y| = 1 \\ \text{and} \\ |{}^xC_i^{k_2}[p+\delta_2]_z - c_z| = 1 \end{array} \right] \right\} $$

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.

where i is the iteration counter, k_1, k_2 indices in candidate sets and δ_1, δ_2 offsets to direct neighbors. We perform several iterations of this process, reducing the number of candidates with every pass. In our current implementation we iterate until having no more than 12 candidates per pixel, which typically requires 2 iterations. If more candidates remain, we keep the first 12 with the smallest crossbar matching error. While it is possible that no candidate coherency exists, this happens rarely in practice.

4.3. Candidate Slab

After candidate generation, we obtain a set of candidate triples at each pixel. These candidates are not only useful for neighborhood matching, but also provide a very good initialization for the synthesis process.

Let us consider a single exemplar. Recall that each candidate triple defines a 3-neighborhood, that is three interleaved $N \times N$ 2D neighborhoods. One 2D neighborhood is in the plane of the exemplar, while the two others are orthogonal to it (see Fig. 5, left) and intersect along a line of N voxels *above* and *below* the exemplar. This provides a way to *thicken* the exemplar and to form *candidate slabs*. To initialize synthesis we create such a slab using the best (first) candidate at each pixel (see Fig. 5, right).

Please note, however, that the slab is formed using a *single* candidate among the several available per exemplar pixel. Using the slab directly as a 3D exemplar would be very limiting: This would ignore all other candidates. Instead, our algorithm exploits the full variety of the candidates for neighborhood matching and uses a slab only for initialization. This is very different from using a 3D exemplar as input, which would require a large example volume to offer a similar variety of candidates.

Figure 5: Left: *Exemplar thickening.* Right: *Candidate slab obtained from the first candidates. The slab is shown from top and bottom view. Notice how coherent structures appear around the exemplar. (This is not a synthesis result - simply a visualization of the candidates).*

5. Lazy Solid Synthesis

For each pixel of each 2D exemplar, we now have a set of pre-computed 3D candidates, which we will use to perform efficient solid synthesis. Our parallel deterministic synthesis is inspired by the 2D parallel algorithm of [LH05]. While it has the same overall structure, it does adapt and extend it in several ways.

5.1. Parallel solid texture synthesis

A first evident change to the original 2D algorithm is that our algorithm performs synthesis in a multi-resolution 3D volume pyramid, instead of operating on a 2D image pyramid. Only part of this volume pyramid may be visited by the algorithm, depending on the subset of desired voxels.

We perform two successive steps at every resolution level: *upsampling* which increases the resolution of the previous level result, and *correction* which applies several passes of neighborhood matching using our pre-computed 3D candidates. Contrary to the original scheme we found it unnecessary to add variation at every level, and perturb the result through *jitter* only once, after initialization. If finer control is desired, jitter could be explicitly added after each upsampling step.

This is summarized below:

```
1. Synthesize( l_start, V_init, E_x,E_y,E_z, xC,yC,zC, DesiredVoxels )
2.     Mask^{l_start} ← ComputeMask(DesiredVoxels, l_start)
3.     V^{l_start} ← tiling of V_init
4.     V^{l_start} ← Jitter(V^{l_start})
5.     For l = l_start ... L
6.         If l > l_start then V^l, Mask^l ← Upsampling(V^{l-1}, Mask^{l-1})
7.         For p = 1...2
8.             V^l, Mask^l ← Correction(V^l, Mask^l, xC,yC,zC)
9. End
```

l_{start} is the level at which synthesis starts. `ComputeMask` computes the mask of voxels that have to be synthesized at level l_{start} (see Sec. 5.2). V_{init} is an initial result from which to begin synthesis, as explained below.

In every voxel of the synthesis pyramid we maintain a coordinate triple, representing a 3-neighborhood. For a voxel at coordinates v in the volume of level l we note the stored triple coordinates $V[v]^l_x$, $V[v]^l_y$ and $V[v]^l_z$. The color of a voxel is obtained by averaging the three colors at coordinates $V[v]^l_x$, $V[v]^l_y$ and $V[v]^l_z$ in respectively E_x, E_y and E_z.

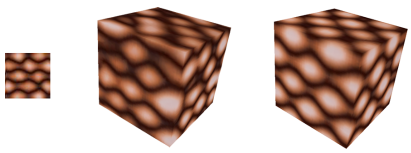


Figure 6: *Quality of solid texture synthesis is improved by using candidate slabs for initialization (right), compared to random initial values (left). This is especially the case on structured exemplars.*

Initialization

To reduce synthesis time, multi-resolution synthesis algorithms can start from an intermediate level of the image pyramid. The initial result given as input is then iteratively refined, with successive passes of neighborhood matching. A good initialization is key to achieve high-quality synthesis.

Since we do not have an example volume, we rely on the candidate slabs (Sec. 4.3) for initialization. They provide a

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.

good approximation of the 3D content to be synthesized. We simply choose one of the candidate slabs as V_{init} and tile it in the volume to initialize synthesis. In practice, we initialize the process three levels above the finest ($l_{start} = L-3$) using the candidate slab from the corresponding level of the exemplar pyramid.

Fig. 6 shows how our initialization improves quality compared to a typical random initialization. In particular, it preserves regularity present in the input images.

Jitter

Since we initialize with a tiling, we have to explicitly introduce variations to generate variety in the result. We hence perturb the initial result by applying a continuous deformation, similar to a random warp. We compute the distorted volume J as:

$$\forall v,\ J[v] = V[v + \sum_{i=0...G} A_i \vec{d}_i\, e^{-\frac{\|v-c_i\|^2}{2\sigma_i^2}}]$$

where v are voxel coordinates, A_i is a scalar, $\vec{d}_i$ a random normalized direction, c_i a random point in space and σ_i controls the influence of Gaussian i. G is typically around 200, with values of A_i ranging from 0.1 to 0.3 and σ_i from 0.01 to 0.05 in unit volume space. While exact values do not matter - the main purpose is to randomize - it is important for A_i to have larger magnitude with smaller σ_i: This adds stronger perturbation at small scales, while adding subtle distortions to coarser scales. Small scale distortions are corrected by synthesis, introducing variety. The overall magnitude of the jitter is directly controllable by the user.

Upsampling

Upsampling is a simple coordinate inheritance: Each of the eight child volume cells inherits three coordinates from its parent, one for each direction. The new coordinates of the child at location ijk within the volume of level l are computed as:

$$\begin{aligned}
V[ijk]^l_x &= 2\cdot V[(\lfloor \tfrac{i}{2}\rfloor, \lfloor \tfrac{j}{2}\rfloor, \lfloor \tfrac{k}{2}\rfloor)]^{l-1}_x + (j \mod 2,\ k \mod 2)\\
V[ijk]^l_y &= 2\cdot V[(\lfloor \tfrac{i}{2}\rfloor, \lfloor \tfrac{j}{2}\rfloor, \lfloor \tfrac{k}{2}\rfloor)]^{l-1}_y + (i \mod 2,\ k \mod 2)\\
V[ijk]^l_z &= 2\cdot V[(\lfloor \tfrac{i}{2}\rfloor, \lfloor \tfrac{j}{2}\rfloor, \lfloor \tfrac{k}{2}\rfloor)]^{l-1}_z + (i \mod 2,\ j \mod 2)
\end{aligned}$$

Correction

The correction step relies on our candidate sets to perform fast neighborhood matching. It is performed on all synthesized voxels simultaneously, in parallel. Input data is read from the previous step result so that neighboring voxels do not influence each other during correction.

The result from the previous step gives us a coordinate triple in each voxel, from which we compute a color by averaging the corresponding three colors from the exemplars. In each synthesized voxel, we start by gathering its *current* 3-neighborhood, that is the one that can be observed in the colored version of the result. We will use this 3-neighborhood to search for a best matching candidate.

Next, we gather a set of potential candidates for the voxel. We visit each of its direct neighbors, and use the stored coordinate triples to gather the candidate sets. This relies on the coherent candidate idea introduced by [Ash01] and [TZL*02]. The final candidate set $\mathscr{C}(v)$ for voxel v is computed as:

$$\begin{aligned}
\mathscr{C}(v) &= \mathscr{C}_x(v) \cup \mathscr{C}_y(v) \cup \mathscr{C}_z(v) \text{ where}\\
\mathscr{C}_x(v) &= \{\, {}^xC^k[V[v+P_x\delta]_x - \delta] : k = 1...K, \delta \in \{-1,0,1\}^2\}\\
\mathscr{C}_y(v) &= \{\, {}^yC^k[V[v+P_y\delta]_y - \delta] : k = 1...K, \delta \in \{-1,0,1\}^2\}\\
\mathscr{C}_z(v) &= \{\, {}^zC^k[V[v+P_z\delta]_z - \delta] : k = 1...K, \delta \in \{-1,0,1\}^2\}
\end{aligned}$$

${}^xC^k[p]$ is the k-th candidate at location p in E_x. P_x, P_y and P_z are 3×2 matrices transforming a 2D offset from exemplar to volume space (see Fig.3).

Each candidate is itself a triple of coordinates forming a 3-neighborhood. We search for the best matching candidate by considering the distance between the candidate 3-neighborhood and the 3-neighborhood we extracted for the current voxel. The distance is a simple L^2 norm on color differences. In practice, we speed up these comparisons using PCA-projected neighborhoods.

We finally replace the coordinate triple in the current voxel by the coordinate triple of the best matching candidate. Note that because the candidate triples have been precomputed and optimized, we are guaranteed that the color disparity between the three colors in each voxel is kept low.

We perform two correction passes at every level of the pyramid, and improve convergence of the correction process by adapting the sub-pass mechanism of [LH05]. We simply perform 8 sub-passes instead of 4 in 2D, processing interleaved subsets of voxels one after the other.

Separation of the first candidate coordinate. While our candidate selection is very efficient, we still end up with many candidate comparisons: We gather 12 candidates from the $3^3 = 27$ direct neighbors (including the center), for a total of 324 candidates per voxel. We further reduce the search space by performing a two step search.

During synthesis, the first step is to search for the best matching 2D candidates in each of the three directions. The second step is to gather the 3D candidates only from these three best matching pixels. This greatly reduces the size of the candidate set to consider, but still allows for a large number of candidate combinations. In practice we keep 4 2D and 12 3D candidates per exemplar pixel at coarse levels, while we reduce to 2 2D and 4 3D candidates at the finest level for maximum performance. At most, we thus perform a search within $27\times 4 = 108$ 2D candidates and $3\times 12 = 36$ 3D candidates.

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.

5.2. Lazy Subset Synthesis

In order to synthesize the smallest number of voxels, we determine, from a requested set of voxels at finest resolution, the entire dependency chain throughout the volume pyramid. This guarantees all necessary information is available to ensure spatial determinism. Fig. 7 illustrates this idea on a simple 2D example.

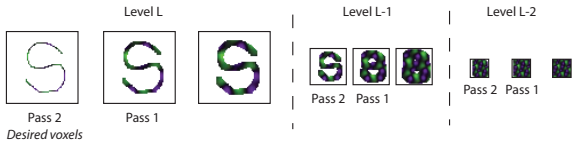


Figure 7: *From a set of desired voxels at finest resolution, we have to determine the entire dependency chain throughout the volume pyramid. The neighborhood matching passes performed at every level imply the dilation of the set of voxels to synthesize.*

To restrict synthesis only on the necessary voxels, we compute a synthesis mask. When $Mask_p^l[v]$ is true, it indicates that voxel v at pass p and level l has to be synthesized. Note that we only need to compute the mask for level l_{start}: During synthesis, the mask for the next level or next pass is trivially obtained through upsampling and correction (see Sec. 5.1).

We compute the mask from from last to first pass, and from fine to coarse levels. The number of required voxels depends on the size and shape of the neighborhoods used during synthesis. In the following pseudo-code, we note $Mask_p^l \otimes NeighborhoodShape$ the dilation of the mask by the shape of the neighborhoods. Function *Downsample* reduces the resolution of the mask and flags a parent voxel as required if any of its children are required. *DesiredVoxels* contains the set of voxels requested by the user.

To compute a single voxel, with $N = 5$, 2 passes and synthesis of the 3 last levels, our scheme requires a dependency chain of 6778 voxels. Note that in a volume the size of the dependency chain grows quadratically with the number of passes.

```
1. ComputeMask (DesiredVoxels)
2.     Mask^{finestlevel}_{lastpass} ← DesiredVoxels
3.     Foreach level l from finest to l_start
4.         Foreach pass p from last to first
5.             Mask^l_{p-1} = Mask^l_p ⊗ NeighborhoodShape
6.         end foreach
7.         If l > l_start then Mask^{l-1}_{lastpass} = Downsample(Mask^l_{firstpass})
8.     end foreach
9.     return Mask^{l_start}
```

6. Implementation and Results

We have implemented our solid texture synthesis approach both entirely in software and using the GPU to accelerate the actual synthesis. All our results are created on an Intel Core2 6400 (2.13GHz) CPU and an NVIDIA GeForce 8800 Ultra.

Note that we sometimes add feature distance [KFCO*07] to the RGB exemplars. Whenever this is the case, the feature distance is shown along with the original image.

6.1. Candidate pre-computation

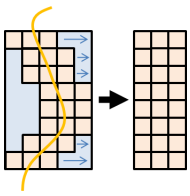
Apart from Fig. 9, all results in the paper are computed from a single example image repeated three times. Depending on the orientation chosen for the image in E_x, E_y and E_z, the pre-computed candidates may be shared. This incurs savings in computation and data structures since we can perform the pre-computation only once. All reported sizes and timings are for a single example image sharing a same candidate data structure, using the orientation depicted in Fig. 3. The entire pre-computation is fast: Typically 7 seconds for 64^2 exemplars, and 25 to 35 seconds for 128^2 exemplars. This includes building the exemplar pyramids, computing the PCA bases and building the candidate sets. Typical memory requirement for our pre-computation data structure is 231KB for a 64^2 exemplar.

After pre-computation we can quickly perform synthesis on any surface, and generate many variations of a same texture as illustrated in the supplemental material. This affords a very convenient tool to decorate objects from a database of pre-computed exemplars.

6.2. GPU implementation

The synthesis algorithm is implemented in fragment shaders, using the OpenGL Shading Language. We unfold volumes in tiled 2D textures, using three 2-channel 16 bit render targets to store the synthesized triples. We pre-compute and reduce the dimensionality of all candidate 3-neighborhoods using PCA, keeping between 12 and 8 dimensions. We typically keep more terms at coarser levels since less variance is captured by the first dimensions. We finally quantize the neighborhoods to 8-bits to reduce bandwidth. Hence, candidate sets are stored in RGBA 8 bit textures.

Synthesizing around surfaces. When texturing objects, our goal is to focus computations only around the voxels intersecting the surface. In order to minimize memory consumption, we perform synthesis into a TileTree data structure [LD07], but other approaches such as octree textures [BD02] could be used. After synthesis, rendering is performed directly using the TileTree, or the texture can be unfolded in a standard UV map.



The TileTree subdivides the surface into a set of square tiles. Each tile is in fact a height-field and corresponds to the set of voxels intersecting the surface. We enlarge the voxel set as described Sec. 5.2, and perform synthesis independently in each tile. In order to reduce the size of the 3D texture used for synthesis, we 'push' the voxel columns at the bottom of the tile, as

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.

illustrated in the inset. This slightly complicates addressing, but greatly reduces memory consumption on the GPU.

Synthesizing tiles independently implies that many voxels at coarse resolution are computed several times. This is especially unfortunate since these voxels only represent 10 percent of the total number of voxels. We reduce this overhead by emulating a very simple cache mechanism for coarse level voxels. We first synthesize a small 32^3 volume up to level $L-1$, and store this intermediate result. When synthesizing the surface, we restart synthesis at level $l_{start} = L-1$ using a tiling of the intermediate result. Variety is added as usual by perturbing the tiling with warping.

When interactively cutting an object, synthesis occurs only once for the newly appearing surfaces. Since the Tile-Tree cannot be updated interactively, we store the result in a 2D texture map for display. For simplicity our implementation only allows planar cuts: The new surfaces are planar and are trivially parameterized onto the 2D texture synthesized when the cut occurs. These textures are shown at the top of the screen in the accompanying video.

6.3. Full volume synthesis and comparisons

While our scheme is designed for fast surface synthesis, we can also use it to quickly generate full volumes of color content. Here we discuss this possibility and use it to compare our algorithm with previous work.

Quality. For comparison purposes, we reproduce in Fig. 8 solid textures similar to those presented in [KFCO*07]. As can be seen, our new approach produces results which are at least of comparable quality and often slightly better. Fig. 9 illustrates how complex structures are synthesized from different example images.

Timings. In Fig. 8, for the 64^2 examples of the first row our method requires a total of 7.22 seconds for synthesizing the 64^3 volume (7 seconds for pre-computation and 220 milliseconds for synthesis). The memory requirement during synthesis is 3.5MB. For the 128^2 image of the last row, our method requires a total of 28.7 seconds for synthesizing the 128^3 volume (27 seconds for pre-computation and 1.7 seconds for synthesis). In comparison, [KFCO*07] reported timings between 10 and 90 minutes.

6.4. Solid synthesis on surfaces

Fig. 10 shows results of synthesizing various solid textures on complex surfaces. Performing synthesis using our lazy scheme results in a very low memory consumption compared to the equivalent volume resolution.

Synthesis speed ranges from 4.1 seconds (dragon) to 17 seconds (complex structure), excluding pre-computation. Storage of the texture data requires between 17.1MB (statue) and 54MB (complex structure), while the equivalent volume resolution is 1024^3 which would require 3GB.

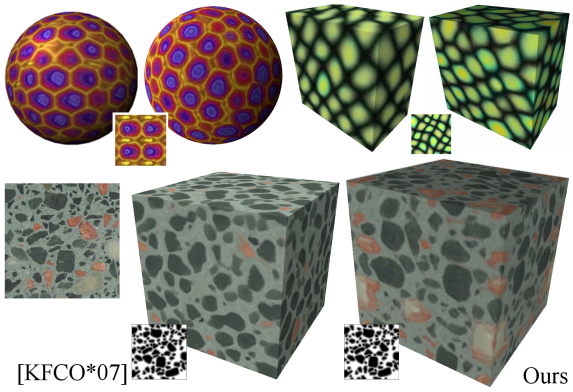


Figure 8: *Comparisons to some of the result textures in [KFCO*07]. For each comparison, our result is shown on the right.*

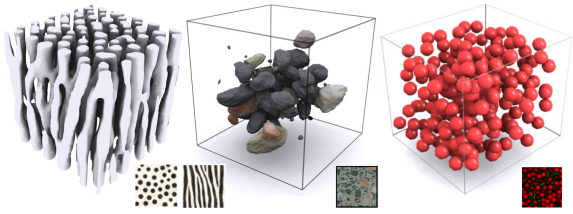


Figure 9: Left: *A volume generated from two different images.* Right: *Transparency reveals shape distribution. Background gray voxels are transparent for the stones, green voxels for the spheres. For clarity we removed shapes crossing the volume boundary.*

While being slower than state-of-the-art pure surface texture synthesis approaches, our scheme inherits all the properties of solid texturing: No distortions due to planar parameterization, a unique feeling of a coherent block of matter, consistent texturing when the object is cut, broken or edited. None of the pure 2D synthesis approaches can enforce these properties easily. Our timings nonetheless allow for on demand synthesis when cutting or breaking objects. Fig. 10 shows four frames of a real-time explosion. The texture has an equivalent resolution of 256^3, while storage requires 1.3MB. The average time for synthesizing a 256^2 texture for a new cut is 8 ms. Please see the accompanying video for several captured interactive cutting sessions. In terms of memory, synthesizing a 256^2 slice of texture content requires 14.4MB. The overhead is due to the necessary padding to ensure spatial determinism (see Sec. 5.2).

Comparison with a simple tiling. A typical approach for solid texturing is to pre-compute a full cyclic volume and to tile it in space for texturing objects. As shown Fig. 11, our scheme offers richer textures than a simple volume tiling and avoids the appearance of repetitive patterns along some particular directions. Recall that synthesis occurs only once: There is no overhead for displaying the object.

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.

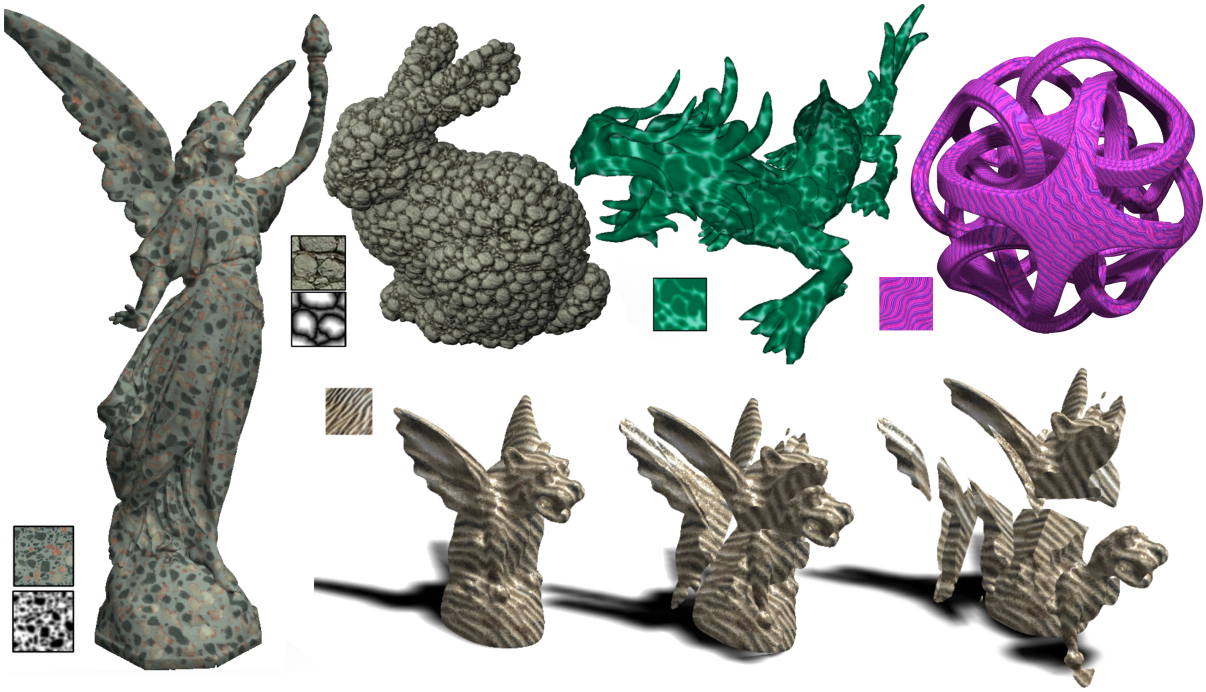


Figure 10: *Results on complex surfaces.* Top row: *Surfaces textured with the equivalent of a* 1024^3 *volume. The bunny is rendered with displacement mapping using synthesized feature distance.* Bottom row: *Three frames of the real-time explosion of a gargoyle model. The interior surfaces are synthesized on demand, in 8 milliseconds on average, while the object explodes. Notice the coherent structures across cut boundaries.*

Comparison with a method using standard 2D candidates. In order to experimentally verify that our 3D candidates provide good quality results with fewer iterations, we also implemented our synthesis algorithm using only standard 2D candidates. As can be seen Fig. 12, it takes roughly twice the number of iterations to obtain a result of equivalent visual quality (we obtain similar numbers on different textures). Due to the increased number of iterations, the size of the dependency chain for computing a single voxel grows from 6778 voxels with 3D candidates to 76812 voxels with 2D candidates, hence a factor of 11.3 in *both* memory usage and speed. This makes local evaluation impractical, and would not be useful for synthesis on surfaces.

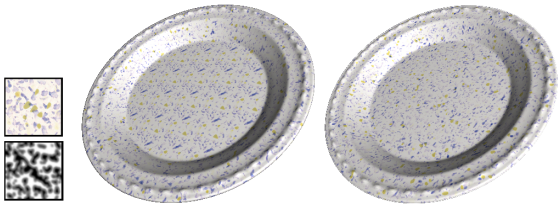


Figure 11: Left: *Result of tiling a* 128^3 *volume to texture a surface. Obvious repetitions are visible along some particular directions.* Right: *Our approach does not exhibit visible repetitions thanks to synthesis.*

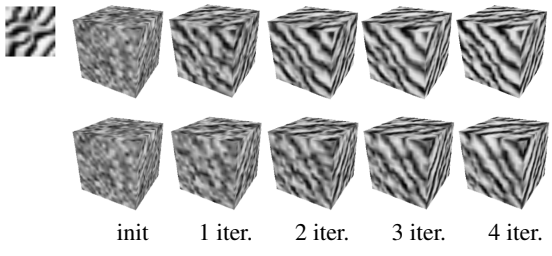


Figure 12: *Comparison of 3D candidates (top) versus 2D candidates (bottom). Using our 3D candidates, a visually pleasing result is reached in 2 iterations. After 4 iterations the 2D candidates barely reached the same quality.*

7. Discussion and Conclusions

Our method is of course not without limitations. In particular, if the exemplars E_x, E_y, E_z do not define a coherent 3D volume, the quality of the result will be poor, as shown Fig. 13 (left). An interesting direction of future research is to exploit our pre-computation to determine whether three exemplars are likely to generate a consistent 3D volume.

A related limitation is that it may be impossible to find coherent candidates during our candidate selection process for some parts of the image. As shown in Fig. 13 (right) this will introduce a bias in the algorithm, removing some features.

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.

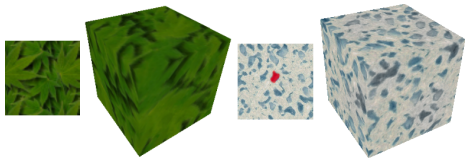


Figure 13: Left: *A case where the 2D example is incompatible with a good solid texture definition.* Right: *A case where a part of the exemplar has no coherent candidates: The red feature is lost through synthesis.*

Conclusion. We have presented a new algorithm for solid synthesis, creating a consistent volume of matter from 2D example images. Our algorithm has the unique ability to synthesize colors for a subset of the voxels, while enforcing spatial determinism. This affords efficient surface synthesis as the complexity in both space and time only depends on the area to be textured.

Our key idea is to pre-compute 3D candidates in a preprocess, by interleaving three 2D neighborhoods from the input images. Thanks to a careful selection, our pre-computed candidates significantly improve synthesis efficiency and reduce the number of iterations required to produce good results. This is key in reducing the size of the dependency chain when evaluating subsets.

Our GPU implementation is fast enough to provide on demand synthesis when interactively cutting or breaking objects, enabling realistic texturing effects in real-time applications and physical simulations.

Acknowledgments

We would like to thank Baining Guo, Li-Yi Wei, and the anonymous reviewers for their help in improving the paper, as well as Su Wang for 3D modeling. This work is partly supported by the NSFC (National Natural Science Foundation of China), grant 10547002.

References

[Ash01] ASHIKHMIN M.: Synthesizing natural textures. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics* (2001), pp. 217–226.

[BD02] BENSON D., DAVIS J.: Octree textures. In *Proceedings of ACM SIGGRAPH* (2002), pp. 785–790.

[DG01] DISCHLER J.-M., GHAZANFARPOUR D.: A survey of 3d texturing. *Computers & Graphics 25*, 10 (2001).

[EF01] EFROS A. A., FREEMAN W. T.: Image quilting for texture synthesis and transfer. *Proceedings of ACM SIGGRAPH* (2001), 341–346.

[EL99] EFROS A. A., LEUNG T. K.: Texture synthesis by non-parametric sampling. In *Proceedings of the IEEE International Conference on Computer Vision* (Corfu, Greece, September 1999), pp. 1033–1038.

[EMP*94] EBERT D., MUSGRAVE K., PEACHEY D., PERLIN K., WORLEY: *Texturing and Modeling: A Procedural Approach*. Academic Press, 1994. ISBN 0-12-228760-6.

[GD95] GHAZANFARPOUR D., DISCHLER J.-M.: Spectral analysis for automatic 3d texture generation. *Computers & Graphics 19*, 3 (1995).

[GD96] GHAZANFARPOUR D., DISCHLER J.-M.: Generation of 3d texture using multiple 2d models analysis. *Computers & Graphics 15*, 3 (1996).

[HB95] HEEGER D. J., BERGEN J. R.: Pyramid-Based texture analysis/synthesis. In *Proceedings of ACM SIGGRAPH* (1995), pp. 229–238.

[HJO*01] HERTZMANN A., JACOBS C. E., OLIVER N., CURLESS B., SALESIN D. H.: Image analogies. In *Proceedings of ACM SIGGRAPH* (2001), pp. 327–340.

[JDR04] JAGNOW R., DORSEY J., RUSHMEIER H.: Stereological techniques for solid textures. *Proceedings of ACM SIGGRAPH* (2004), 329–335.

[KEBK05] KWATRA V., ESSA I., BOBICK A., KWATRA N.: Texture optimization for example-based synthesis. In *Proceedings of ACM SIGGRAPH* (2005), pp. 795–802.

[KFCO*07] KOPF J., FU C.-W., COHEN-OR D., DEUSSEN O., LISCHINSKI D., WONG T.-T.: Solid texture synthesis from 2d exemplars. In *Proceedings of ACM SIGGRAPH* (2007).

[KSE*03] KWATRA V., SCHÖDL A., ESSA I., TURK G., BOBICK A.: Graphcut textures: Image and video synthesis using graph cuts. *Proceedings of ACM SIGGRAPH* (2003), 277–286.

[LD07] LEFEBVRE S., DACHSBACHER C.: Tiletrees. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2007).

[LH05] LEFEBVRE S., HOPPE H.: Parallel controllable texture synthesis. *Proceedings of ACM SIGGRAPH 24*, 3 (2005), 777–786.

[MA97] MOUNT D., ARYA S.:, 1997. ANN: A library for approximate nearest neighbor searching. CGC 2nd Annual Fall Workshop on Computational Geometry, http://www.cs.umd.edu/~mount/ANN, 1997.

[ONOI04] OWADA S., NIELSEN F., OKABE M., IGARASHI T.: Volumetric illustration: Designing 3d models with internal textures. *Proceedings of ACM SIGGRAPH* (2004), 322–328.

[QY07] QIN X., YANG Y.-H.: Aura 3d textures. *IEEE Transactions on Visualization and Computer Graphics 13*, 2 (2007), 379–389.

[TZL*02] TONG X., ZHANG J., LIU L., WANG X., GUO B., SHUM H.-Y.: Synthesis of bidirectional texture functions on arbitrary surfaces. In *Proceedings of ACM SIGGRAPH* (2002), pp. 665–672.

[Wei02] WEI L.-Y.: *Texture synthesis by fixed neighborhood searching*. PhD thesis, 2002. Stanford University, Advisor-Marc Levoy.

[WL00] WEI L.-Y., LEVOY M.: Fast texture synthesis using tree-structured vector quantization. In *Proceedings of ACM SIGGRAPH* (2000), pp. 479–488.

[WL03] WEI L.-Y., LEVOY M.: Order-independent texture synthesis, 2003. Technical Report TR-2002-01, Computer Science Department, Stanford University.

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.